Süsteemprogrammeerimine keeles C

Loeng 12

Milles räägitakse programmi argumentide töötlusest, mälukasutusest sest kordamine on tarkus ning veel kahest mugavusfunktsioonist

Eksamid

(Traditsiooniliselt esmaspäeviti kell kümme)

11. ja 18 jaanuar?

Eelmises osas (Lõim)

Lõim on n.ö. kergprotsess või *light weight process*, nagu seda aegajalt kutsutakse.

Ühe protsessi raames saab samasse virtuaalsesse mäluaadresside ruumi tekitada mitu käivituskonteksti, mis kordamööda või paralleelselt jooksevad.

Protsess omab ressursse – lõimel on enda omad ainult oma stack ja mõned lõimepõhised muutujad

Eelmises osas (Eelised)

Ümberlülitamine on tihti kiirem ning suhtlemisele kulub vähem ressurssi.

Kõik eelised, mis suudad välja mõelda ühtse aadressruumi korral võrreldes eraldatud aadressruumiga.

Parem programmistruktuur

Tuuma (*kernel*) teadlikkuse korral saab lõimesid panna eraldi protsessoritesse

Eelmises osas (POSIX & SOLARIS)

Kaks lõimede librat, mis meid huvitavad

POSIX lõimed (Pthreads)

SOLARIS lõimed

Eelmises osas (Lõime loomine)

phtread_create() loob etteantud funktsioonist lõime, mida eraldiseisvalt käivitama hakatakse

Lõim seiskub kui:
- lõimena käivitatud funktsioon tagastub
- lõim kutsub välja pthread_exit() funktsiooni
- keegi samas protsessis kutsub välja exit()
- kaaslõim kutsub lõime ID-ga välja pthread_cancel()

Eelmises osas (Niitmine)

Lõpetanud tavalised lõimed jäävad lõppedes vaheolekusse (analoog zombiprotsessiga), kus on võimalik lugeda nende tulemus phtread_join() abil.

Lõim on võimalik viia *detached* seisundisse pthread_detach() käsuga, mis "ühendab nad lahti" ning tähendab, et nad lõpetamisel ka tegelikult hävivad.

Reeglina on kasutusel *detached* lõimed.

Eelmises osas (jagatud muutujad)

Suur lõimede eelis ja nuhtlus, sest tekitab järjekordselt palju "huvitavaid" vigu.

Jagatud on globaalsed muutujad (kogu *heap*).

Stack ei ole jagatud, kuid pole ka kaitstud kirjutamise eest.

Funktsioonide staatilised muutujad on jagatud.

Eelmises osas (Ralli)

Kui mõne jagatud muutuja kirjutamine või lugemine ei ole atomaarne tegevus, võib juhtuda, et teine lõim muudab selle poole kirjutamise pealt ära. *Race condition.*

Eelmises osas (Mutex)

Mutex on muutuja, mille abil saab vältida jagatud mälusegmentidega rallitamist.

Mutexit saab luua, lukustada ja avada (init, lock, unlock)

pthread_mutex_init(), pthread_mutex_lock(), pthread_mutex_unlock()

Üks mutex on korraga ainult ühe lõime käes

Eelmises osas (Mutexi käitumine)

Mutex = MUTual EXclusion device

Kui lõim tahab mutexit omale on variandid
- mutex vaba: antakse lukustus
- mutex hõivatud: lõim blokib kuni mutex vabaneb

Deadlocki võimalus

Eelmises osas (Tingimusmuutujad)

Condition variable; võimaldavad oodata mingi tingimuse täitumist.

Pannakse ühe konkreetse mutexi külge.

Põhitegevused: wait, signal

Wait vabastab seotud mutexi ning jääb ootele kuni pasundatakse, et hei, midagi toimus, antakse mutex lõimele tagasi ning ootamine lõppeb.

Tänases osas

- Programmiargumentide inimlik töötlus
- Mäluhaldusest
- system()
- error()

Boonus: Semaforid

- Semaforid on atomaarse ligipääsuga loendurid, mis võimaldavad lõimedel omavahel ressursiarvestust pidada.
- Prototüübid semaphore.h failis
- Tegevused: Atomaarne suurendamine, Ootamine kuni on suurem kui 0 ning seejärel kahandamine.
- Võib kasutada näiteks järjekorras olevate päringute arvestamisel vms

Programmi käsureavalikud

- Kokkuleppeliselt tähistatakse programmi valikud käsureal miinusmärgiga (-) ja ühe tähega (ei ole päris tõsi, aga arvake praegu et on)
- Valikuid võib üksteise otsa kirjutada (kui neil pole parameetreid)
- Valikutel võivad olla parameetreid

```
myprog -a -b
myprog -ab
myprog -ba

myprog2 -i infile -ooutfile
```

- Kuidas sellises situatsioonis laisk olla?

getopt()

```
#include <unistd.h>
int opterr;
int optopt;
int optind;
char * optarg;
int getopt(int argc, char**argv, const char *options)
```

- Tagastab valikute nimekirjast järgmise
- options näitab ära võimalikud "lubatud" käsureavalikud, täht võimaliku tähemärgi, sellele järgnev koolon (:), et järgneb argument, ja (::) kui argument pole kohustuslik
- opterr: kui määrata 0, ei näita vigu välja
- optind: järgmine valik

getopt() (2)

- getopt() vahetab tavaolukorras argv sees olevate valikute järjekorda kuni kõik miinuseta on lõpus
- Kui rohkem valikuid pole, tagastub -1
- Valikute lõppemisel saab kontrollida üle jäänud argumente võrreldes argc-d ja optind muutujat.
- Tundmatu valiku puhul tagastub '?', tundmatu märk salvestatakse muutujasse optopt

```
getopt(argc, argv, "abc:");
```

getopt() (3)

- Reeglina kutsutakse välja tsüklis, mis lõpetab, kui getopt() tagastab -1
- Tagastatavat väärtust valitakse enamasti switch-lausega, kus väärtustatakse iga valiku puhul muutuja, mis hiljem programmi tööd vastavalt mõjutab
- Ülejäänud programmiargumentide jaoks kasutatakse teist tsüklit

Näide

```
#include <unistd.h>
#include <stdio.h>

int
main (int argc, char **argv)
{
  int aflag = 0;
  int bflag = 0;
  char *cvalue = NULL;
  int index;
  int c;

  opterr = 0;

  while ((c = getopt (argc, argv, "abc::")) != -1)
```

Näide (2)

```
 switch (c) {
  case 'a':
  aflag = 1;
  break;
 case 'b':
  bflag = 1;
  break;
 case 'c':
  cvalue = optarg;
  break;
 case '?':
    if (isprint (optopt))
     fprintf (stderr, "Unknown option `-%c'.\n", optopt);
    else
     fprintf (stderr, "Unknown option char `\\x%x'.\n",
                  optopt);
    return 1;
 default:
  abort();
}
```

Näide (3)

```
    printf ("aflag = %d, bflag = %d, cvalue = %s\n",
            aflag, bflag, cvalue);

    for (index = optind; index < argc; index++)
      printf ("Non-option argument %s\n", argv[index]);
    return 0;
}
```

GNU täiendused

- GNU projekti raames on libc-s ka getopt_long
- Selle eesmärgiks on programmide õppimine lihtsamaks teha:

```
ls -Ah
ls --almost-all --human-readable
```

- Pikad valikud algavad topeltmiinusega (--)
- Pika valiku argumendid on kujul:

  --valikunimi=väärtus

- unistd.h asemel on prototüüp eraldi getopt.h failis

struct option

- getopt_long() võtab argumendiks massiivi valikuid kirjeldavatest structidest

```
struct option {
    const char *name;
    int has_arg;
    int *flag;
    int val;
}
```

- name: valiku nimi
- has_args: no_argument, required_argument või optional_argument
- flag ja val osas juurdleme põhjalikumalt kohe

struct option (2)

- flag:
  - kui on pointer mingile väärtusele, salvestatakse **var** sisu näidatud aadressile, kui valik peaks ette juhtuma
  - kui on 0, on **var** mingi unikaalne number, mis valikut tuvastab (enamasti sama, mis oleks tavalise getopti puhul; pärast struct optioni näidet selgub, miks see hästi loogiline on)
- Massiivi viimane element peab olema struct option, mille kõik elemendid on 0

struct option-i näide

```
static struct option long_options[] = {
   /* siin on flag määratud. */
     {"lamisejа", no_argument, &verbose_flag, 1},
     {"tasane",   no_argument, &verbose_flag, 0},
   /* siin on flag 0 .
      eristamine käib tähtede/järjekorranr-i abil. */
     {"add",      no_argument,        0, 'a'},
     {"append",   no_argument,        0, 'b'},
     {"delete",   required_argument, 0, 'd'},
     {"create",   required_argument, 0, 'c'},
     {"file",     required_argument, 0, 'f'},
     {0, 0, 0, 0}
};
```

getopt_long()

- Töötab lühikestega täpselt nagu tavaline getopt()
- Pikkadega sõltub valiku flag parameetrist: kas salvestab kuskile muutuja või tagastab väärtuse val
  - seega kui tagastame lühikese valiku tähe, saame kasutada ka pika puhul sama käitumist

```
int getopt_long (int argc, char *const *argv,
                 const char *shortopts,
                 const struct option *longopts,
                 int *indexptr)
```

- Kui ülaltoodust ei piisa, on olemas argp(), mis meenutab karujahti keskmaaraketiga: väga tõhus, aga sihtimine on veidi keeruline

getopt_long() näidis

```
c = getopt_long (argc, argv, "abc:d:f:",
                 long_options, &option_index);
    if (c == -1) /* valikute ots */
        break;
    switch (c) {
      case 0:  /* kui lipp, ignoreerime praegu. */
        if (long_options[option_index].flag != 0)
                 break;
        printf ("option %s",
                   long_options[option_index].name);
        if (optarg)
            printf (" with arg %s", optarg);
        printf ("\n");
        break;
      case 'a':
        puts ("option -a\n");
        break;
      case 'b':
        /* ... */
```

Getopt lõpetuseks

- getopt() on üsna lihtne, mugav ja veakindel
- getopt_long() on tunduvalt keerulisem, kuid on ikkagi mugavam, kui ise analoogne parsija kirjutada
- Kasutage!

Mäluhaldusest uuesti

- Reeglina saab protsess suure virtuaalse aadressruumi (aadress 0 kuni aadress "vägasuur")
- Ei pruugi olla jätkuv: igale aadressile ei saa andmeid salvestada
- Jaguneb *page*ideks. (tüüpiliselt mahuga 4kB)
- Iga *page* elab kas päris mälus (*frame*) või mõnes muus kohas (kettapuhvris)
  - Tühi mälu märgitakse lihtsalt, et "seal kõik nullid"
- Virtuaalsed aadressid ühendatakse reaalsete *frame*ide või selle muu kohaga

Page fault

- Kuna virtuaalmälu on rohkem kui päris mälu, tuleb *page*sid pärismälu ja tagavaramälu vahel vahetada
- Tegevus nimega *paging*
- *Page fault*: katse lugeda *page*i, millel puudub vaste reaalses mälus
- Kui toimub, loetakse leht tagavaramälust "päris" mällu: paari millisekundine operatsioon võtab järsku suurusjärgu võrra kauem aega
- Kui "ketas ragistab", on palju *page fault*e

Segmendid

- Töötav programm paikneb mälus kolmes segmendis:
  - *text segment*: programmitekst e. käivitatav kood.
  - *data segment*: programmi andmete segment, exec hõivab seda ette, saab ise juurde võtta-
  - *stack segment*: programmi *stack*; kasvab vajadusel, ei kahane. (Ehk siis: segment ei kahane, *stack* ise võib kahaneda.)

Viisid mälu saamiseks

- Mälu saamiseks on kaks viisi
  - käivitades (exec) kui programm mällu loetakse eraldatatakse mälu programmitekstile, konstantidele ja staatilistena deklareeritud muutujatele
  - programselt:
    - automaatsed muutujad
    - malloc
    - mmap: faili virtuaalse mäluga vastavusse seadmine
  - fork: *copy on write trikk*
- Programmi lõpetamisel mälu eraldi ei vabastata: kogu aadressruum ära

Staatilised ja Automaatsed muutujad

- Staatiline hõivamine: Globaalmuutujad ja *static* tüüpi andmed. Hõivatakse programmi käivitumisel ja ei vabastata kunagi
- Automaatne hõivamine: Juhtub siis, kui deklareerida automaatne muutuja nagu funktsiooniargument või kohalik muutuja. Hõivatakse mälu deklareerivasse blokki sisenedes ning vabastatakse sealt väljumisel

- Võimalik C keele enda abil

Dünaamiline hõivamine

- Librade abil saab ise mälu juurde küsida.
  - malloc(), realloc(), calloc()
- Ühtki muutujat ei saa kunagi hoida dünaamilises mälus (seepärast kasutame pointereid)
- GNU malloc
  - ei fragmenteeri mälu (kõrvalseisvad vabad tükid ühendatakse probleemideta)
  - väga suured (kõvasti suuremad kui *page*) hõivatakse mmap() abil:
    - läve määramine funktsiooniga mallopt()

Mäluhalduse jälgimine

- Funktsioon ja lõbusa nimega funktsioon:

```
void mtrace(void);
void muntrace(void);
```

- Salvestavad keskkonnamuutujaga MALLOC_TRACE määratud nimega faili mälukasutusstatistika mälu hõivamiste ja vabastamiste kohta.
- Esimene aktiveerib, teine deaktiveerib seire
- GNU spetsiifiline: mcheck.h failist saab
- Fail ei ole inimloetav. Arusaamiseks programm:

```
mtrace programminimi mtrace-log
```

mmap()

- mmap() poogib faili loetavasse virtuaalmällu (või veeb seda anonüümselt)
- Kohati tõhusam:
  - Loeme mällu ainult jupid, mida ka reaalselt kasutame
  - mmap() asjad saab vajadusel kettale tagasi kirjutada
  - saame avada faile, mis on suuremad kui mem+swap

```
void * mmap (void *address, size_t length, int protect,
             int flags, int filedes, off_t offset)
```

  - Parameetrid: aadress kuhu soovime mappingut, pikkus, kuidas kaitsta, kuidas hallata, failideskriptor ja millisest faili punktist alustada

mmap() parameetritest

- prot: PROT_READ, PROT_WRITE, PROT_EXEC bitid
  - sõltuvalt süsteemist võib esineda anomaaliaid: *write* on enamasti ka *read* või kirjutuskaitstud failidesse ei saa kirjutada ka siis, kui PROT_READ puudub
- flags: *mappingu* olemusest:
  - MAP_PRIVATE: faili tagasi ei kirjutata, muutmisel kirjutatakse tavamällu ja hallatakse sõltumatult
  - MAP_SHARED: muutused kajastuvad ka failis & teiste protsesside jaoks
  - MAP_FIXED: nõua kindel aadress või ebaõnnestu
  - MAP_ANONYMOUS: ära seo failiga (mõni süsteem küsib täpselt nii heap-i juurde)

munmap() & msync() & madvise()

- munmap(): Eemaldab *memory mapping*u etteantud aadressist etteantud aadressini (kasvõi mitu tükki korraga); võib sisaldada ka *mapping*uta lõike.
- msync(): Kirjutab *mapping*u etteantud aadressist etteantud mahus faili.
- madvise(): selgitab kernelile aadressivahemiku mappingu iseloomu: kas kasutame juhupöördumist, loeme järjest, läheb igal juhul kõike vaja, või ei vaja enam üldse ja ta võib täiesti kõik seal sees ära unustada ja klient ka ei lahku toast hüsteeriliselt karvu katkudes

system()

- fork() ja exec() asemel on mõnikord lihtsam kasutada:

```
int system(const char *command);
```

- system () käivitab shelli /bin/sh -c väljakutsega
- Tagastab käivitatud programmi tagastusväärtuse või -1 või 127 vea korral

error()

- fprintf(stderr, "viga\n") asemel:

```
void error(int status, int errnum,
 const char *format, ...);
```

- Trükib stdout-i programminime, kooloni ja teie kirjutatud veateate
- Kui errnum pole null, trükib veel ühe kooloni ja veateate, mille annaks perror(errnum)

Lahkumine